Relazione sul Progetto Homework 2

Indice

Indice figure

Indice tabelle

I. Introduzione

La consegna richiede di estendere il sistema precedente sviluppato nell'homework1 migliorando la gestione delle operazioni di lettura/scrittura sul database con il pattern CQRS e implementando un sistema asincrono di notifiche per inviare un'e-mail o notifiche Telegram se correttamente configurato dall'utente. Gli utenti potranno definire soglie personalizzate (high-value e low-value) per i ticker preferiti, configurabili in fase di registrazione o tramite operazioni. Il database verrà aggiornato per supportare questi nuovi parametri. Il DataCollector notificherà il completamento dell'aggiornamento dei valori dei ticker inviando un messaggio su un topic Kafka (to-alert-system). Un nuovo servizio, l'AlertSystem, analizzerà il database e invierà notifiche su un altro topic Kafka (to-notifier) per gli utenti i cui ticker hanno superato le soglie definite. Infine, l'AlertNotifierSystem invierà agli utenti le notifiche tramite e-mail e qualora sia presente anche una notifica Telegram. Introdurre Kafka in questo contesto è fondamentale per garantire un sistema asincrono, scalabile e resiliente. Questo approccio garantisce un processo scalabile e asincrono per la gestione delle soglie e delle notifiche. Le modifiche apportate al sistema verranno mostrate in dettaglio, evidenziando modifiche alle scelte progettuali, modifiche al database, logica di gestione delle notifiche, integrazione con il sistema Kafka e comunicazione gRPC.

Implementazione del Pattern CQRS

Per migliorare la gestione delle operazioni di lettura e scrittura sul database, è stato introdotto il pattern **CQRS** (Command Query Responsibility Segregation), separando le operazioni in tre flussi distinti:

- **Command**: gestisce le operazioni che modificano lo stato del sistema (registrazione di nuovi utenti, aggiornamento dei profili).
- **Lecture**: si occupa delle operazioni di lettura, come la consultazione dei valori dei ticker e delle soglie associate agli utenti.
- **Connection:** gestisce la connessione del database evitando di ripeterlo in ogni file.

L'introduzione di CQRS migliora le prestazioni del sistema e ottimizza le operazioni di lettura, separando le preoccupazioni e facilitando la gestione dei dati in modo scalabile.

Sistema di Notifica Asincrono con Kafka

Una delle richieste principali del progetto è stata l'implementazione di un sistema di notifiche che avvisa l'utente quando il valore di un ticker scende sotto o supera la soglia definita. Questo sistema è stato realizzato tramite l'uso di Kafka, un sistema di messaggistica asincrona. Grazie alla sua architettura, ci permette di trasferire i messaggi tra i vari componenti del sistema in modo rapido e affidabile. Inoltre, l'uso di Kafka ci consente di separare logicamente i produttori e i consumatori di dati, senza preoccuparsi direttamente di chi li utilizzerà o di come saranno elaborati. Questo approccio migliora la modularità del sistema e lo rende più flessibile. Un altro vantaggio importante è la persistenza dei messaggi. Anche in caso di guasti o interruzioni, Kafka garantisce che i messaggi non vadano persi e possano essere recuperati in un secondo momento. Questo aumenta l'affidabilità complessiva del sistema. Inoltre, scegliendo Kafka, ci prepariamo a gestire scenari futuri in cui il carico di lavoro o le necessità del sistema potrebbero aumentare. Kafka, infatti, è

altamente scalabile e può crescere con il sistema, aggiungendo nuovi broker, partizioni e funzionalità senza richiedere modifiche radicali.

Viene deciso di utilizzare un solo broker Kafka per diverse ragioni. La prima è legata alla semplicità: implementare un singolo broker è più facile e richiede meno configurazioni rispetto a un intero cluster. Questo approccio è particolarmente utile, infatti il carico di lavoro è basso e non giustifica la complessità di un cluster con più broker. Inoltre, con un solo broker, possiamo ridurre l'uso di risorse hardware e semplificando la manutenzione. Questo ci consente di bilanciare la necessità di semplicità operativa con la possibilità di espansione futura, costruendo una base solida per il sistema senza complicare eccessivamente l'implementazione iniziale. Essendo anche tutto il sistema mantenuto in locale.

Vediamo come funziona Kafka:

- **DataCollector**: Il **DataCollector** è responsabile del recupero dei valori aggiornati dei ticker e dell'aggiornamento nel database. Dopo l'aggiornamento, invia un messaggio (producer) a un topic Kafka chiamato **to-alert-system**, notificando che l'aggiornamento è stato completato.
- **AlertSystem**: L'**AlertSystem** è un servizio che ascolta il topic **to-alert-system,** quindi consumatore e produttore del topic **to-notifier**. Una volta ricevuto un messaggio, il sistema controlla se i valori dei ticker sono al di sopra del **high-value** o al di sotto del **low-value**. Se una soglia è superata, l'AlertSystem invia un messaggio al topic **to-notifier** con le informazioni necessarie per inviare la notifica (e-mail, chat-id, ticker, condizione di superamento della soglia).
- **AlertNotifierSystem**: L'**AlertNotifierSystem** è un consumatore del topic **to-notifier**. Una volta ricevuto il messaggio, invia un'e-mail all'utente, utilizzando le informazioni fornite (e-mail, ticker, soglia superata). Se il valore del ticker supera la soglia superiore, l'e-mail avvisa che "Il ticker ha superato la soglia superiore", mentre se scende sotto la soglia inferiore, l'e-mail avvisa che "Il ticker è sceso sotto la soglia inferiore

Oltre all'invio dell'e-mail se correttamente configurato è possibile inviare una notifica su Telegram con le stesse informazioni.

II. Schema Architetturale

Il sistema descritto si basa su un'architettura a microservizi, progettata per favorire modularità, scalabilità e manutenibilità.

Composizione dell'Architettura

Verranno presentati solo i microservizi aggiunti dal precedente homework.

Zookeeper:

- **Funzione**: Zookeeper è utilizzato per la gestione dei broker Kafka. Agisce come sistema di coordinamento e gestione dei metadati per il Kafka cluster, assicurando che i broker Kafka possano operare correttamente in un ambiente distribuito.

- **Ruolo nell'architettura**: Gestisce la configurazione di Kafka e coordina la comunicazione tra i broker per garantire l'affidabilità e la coerenza del sistema.

Schema Registry:

- **Funzione**: Lo Schema Registry gestisce gli schemi dei dati scambiati tra i microservizi, in particolare tra il sistema Kafka e gli altri componenti. Fornisce una gestione centralizzata degli schemi per i messaggi Kafka, garantendo che i dati inviati e ricevuti siano conformi agli schemi definiti.
- **Ruolo nell'architettura**: Assicura che i dati scambiati tra i microservizi siano validi, prevenendo errori di serializzazione e garantendo la compatibilità tra le diverse versioni degli schemi.

Kafka Broker (broker1):

- **Funzione**: Kafka funge da sistema di messaggistica asincrona. I broker Kafka sono responsabili della gestione dei topic e dei messaggi, facilitando la comunicazione tra i microservizi in modo altamente scalabile e resiliente.
- **Ruolo nell'architettura**: Kafka è utilizzato per orchestrare la comunicazione asincrona tra i vari microservizi, consentendo l'aggiornamento e la notifica degli utenti tramite flussi di dati asincroni (event-driven).

Telegram Bot Service:

- **Funzione**: Questo servizio implementa un bot Telegram che interagisce con gli utenti per associare un'e-mail al loro `chat_id`, salvando queste informazioni nel database.
- **Ruolo nell'architettura**:
  - **Gestione delle richieste utenti**: Riceve messaggi dagli utenti tramite l'API di Telegram e li elabora per raccogliere dati.
  - **Validazione dei dati**: Controlla se il `chat_id` di un utente è già associato a un'email nel database.
  - **Persistenza dati**: Salva le associazioni `e-mail-chat_id` nel database.
  - **Comunicazione**: Invia messaggi agli utenti per guidarli nel completamento della configurazione.

Alert System Microservice:

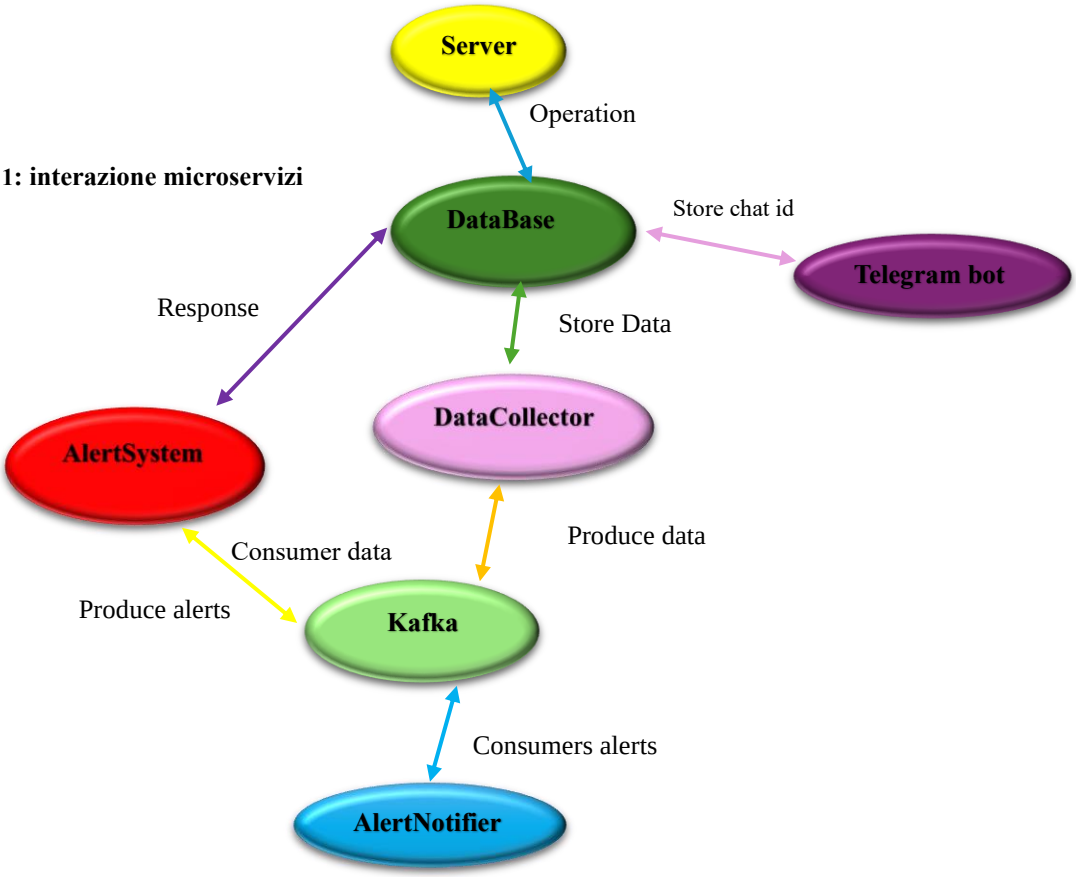
- **Funzione**: Questo microservizio si occupa di monitorare i dati aggiornati dei tickers e di determinare se un'azione deve essere intrapresa in base alle soglie configurate per ogni utente.
- **Ruolo nell'architettura**:
  - **Monitoraggio delle soglie**: Verifica se il valore di un ticker supera la soglia massima o scende sotto la soglia minima.
  - **Generazione di eventi di alert**: Quando una soglia viene superata, invia un messaggio al sistema di notifica per informare gli utenti.

Alert Notifier System Microservice:

- **Funzione**: Il sistema di notifiche gestisce l'invio delle notifiche via e-mail o su altre piattaforme agli utenti quando le condizioni di alert sono soddisfatte.
- **Ruolo nell'architettura**: Riceve eventi dal sistema di alert e invia notifiche agli utenti per informarli dei cambiamenti nel valore dei loro tickers preferiti.

Per motivi di semplificazione grafica e per garantire una visualizzazione più chiara e intuitiva, tutte le componenti di Kafka, tra cui Zookeeper, Schema Registry e il broker Kafka (Broker1), sono state raggruppate all'interno di un unico riquadro denominato 'Kafka'.

Figura 1: interazione microservizi



III. Interazioni tra i Componenti da sistemare

DataCollector ↔ Kafka:

Descrizione:

- **DataCollector** raccoglie dati finanziari utilizzando la libreria **yfinance**. Per ogni ticker (titolo azionario), esegue una richiesta per ottenere l'ultimo valore disponibile.
- Una volta raccolto il dato, il **DataCollector** invia un messaggio notificando l'aggiornamento a un **topic Kafka** specifico. Nel caso specifico, i dati vengono inviati al topic **to-alert-system**.

AlterSystem ↔ Database:

Descrizione:

- L'**AlterSystem** interroga il **Database** per ottenere ulteriori informazioni sui ticker come recuperare i valori minimi, massimi.
- Il Database risponde con i dati necessari per completare l'elaborazione dell'**AlterSystem**, che poi procederà a generare un messaggio di allerta se vengono soddisfatte le condizioni predefinite.

AlterSystem ↔ Kafka:

Descrizione:

- Dopo aver elaborato le informazioni e verificato le condizioni, l'**AlterSystem** produce messaggi di allerta e li invia a un altro **topic Kafka** (to-notifier).
- **Kafka** riceve i messaggi dal **DataCollector** e li rende disponibili nei suoi topic. L'**AlterSystem** consuma questi messaggi dal topic **to-alert-system** e li elabora.
- Durante l'elaborazione, l'**AlterSystem** esegue ulteriori controlli tramite il Database per determinare se i dati ricevuti soddisfano determinati criteri (ad esempio, se un ticker ha raggiunto un valore che richiede una notifica).
- Una volta verificato che le condizioni siano soddisfatte, l'**AlterSystem** invia il messaggio di allerta al topic **to-notifier**, utilizzando un altro produttore Kafka.

Kafka ↔ AlertNotifierSystem:

Descrizione:

- **AlertNotifierSystem** consuma i messaggi di allerta generati dall'**AlterSystem** e disponibili su Kafka.
- Questi messaggi vengono processati dall'**AlertNotifierSystem** per creare notifiche indirizzate all'utente. L'**AlertNotifierSystem** può inviare notifiche via **e-mail** o **Telegram**.

Telegram bot↔ database:

Il bot ha un'interazione chiave con il database per memorizzare e validare le informazioni degli utenti. Le azioni specifiche sono:

- **Validazione Dati**: Verifica se il chat_id è già associato a un'email nel database. Se sì, informa l'utente che l'associazione esiste già.
- **Persistenza Dati**: Salva l'associazione (chat_id-email) nel database se non esiste già. Questo consente di inviare notifiche future.
- **Operazioni Database**: Esegue lettura (per verificare associazione) e scrittura (per memorizzare nuova associazione).

Telegram bot↔ Client:

Il bot interagisce direttamente con l'utente tramite l'API di Telegram, guidando il processo di raccolta e gestione delle informazioni. Le azioni specifiche verso l'utente sono:

- Gestione delle Richieste: Riceve chat_id e e-mail dagli utenti per configurare l'account.
- Comunicazione: Invia messaggi di conferma, errore o guida:
    - Conferma se l'associazione è riuscita.
    - Notifica se l'e-mail è già associata al chat_id.
    - Guida l'utente se l'e-mail è errata o se è necessario completare l'associazione.

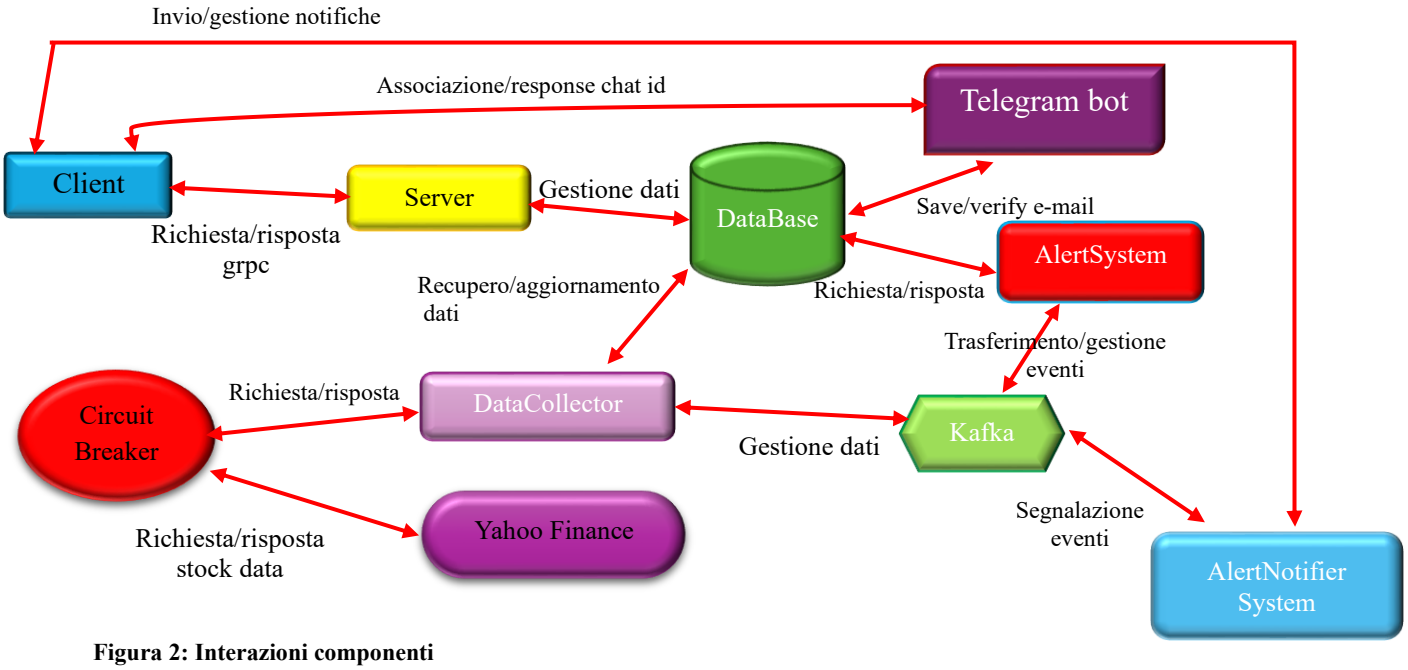


Figura 2: Interazioni componenti

IV. Modifiche apportate rispetto all'homework 1

Per supportare i nuovi requisiti, è stato necessario aggiornare la struttura del database. In particolare, è stata modificata la tabella **UserTickers**, che gestisce la relazione tra gli utenti e i loro ticker preferiti. Sono stati aggiunti due nuovi campi:

- **high_value**: rappresenta la soglia superiore. Se il valore del ticker supera questa soglia, l'utente riceve una notifica.
- **low_value**: rappresenta la soglia inferiore. Se il valore del ticker scende sotto questa soglia, l'utente riceve una notifica.

Durante la fase di registrazione, l'utente può scegliere di fornire i valori. È stata implementata una validazione che assicura che, se entrambi i valori sono forniti, **high_value** sia maggiore di **low_value**.

V. Modifiche Operazioni gRPC

Per permettere agli utenti di modificare i valori **high_value** e **low_value**, sono state implementate nuove operazioni gRPC. In particolare, la comunicazione gRPC è stata modificata per includere i

nuovi campi **max_value** e **min_value** per la registrazione e l'aggiornamento dei ticker, nonché per introdurre il nuovo metodo **UpdateMinMaxValue**, che consente di aggiornare le soglie per i ticker esistenti.

Le modifiche principali alla comunicazione gRPC includono:

- **Registrazione dell'utente**: L'utente fornisce i propri dati, inclusi il ticker e, facoltativamente, i valori **high-value** e **low-value**.
- **Aggiornamento del profilo**: Gli utenti possono aggiornare uno o entrambi i valori tramite una chiamata gRPC, con il sistema che verifica che **high-value** sia maggiore di **low-value** se entrambi sono presenti.

Queste modifiche alla comunicazione gRPC sono state implementate per supportare le nuove funzionalità di gestione delle soglie e garantire che le modifiche vengano correttamente riflesse nel database. Le nuove operazioni gRPC consentono agli utenti di aggiornare i propri ticker e le soglie in modo semplice e diretto.

Modifiche API Implementate

Nell'implementazione del secondo homework è stato necessario inserire delle modifiche nelle precedenti api utilizzate. Di seguito Verranno descritte nel dettaglio le api in cui è stato necessario introdurre delle modifiche.

1. **RegisterUser**
    - **Descrizione**: Permette di registrare un nuovo utente, fornendo la propria e-mail, un ticker che deve essere valido (presente nel file CSV) e i valori massimo e minimo per il ticker introdotti in questo homework. Il sistema restituisce un messaggio di successo o errore, con un valore booleano che indica se la registrazione è avvenuta con successo.
2. **AddTickerUtente**
    - **Descrizione**: Permette all'utente di aggiungere un nuovo ticker. Il ticker deve essere valido (presente nel file CSV) e introdotto la possibilità di fornire i valori massimo e minimo per il ticker. Il sistema aggiunge il ticker al database e restituisce un messaggio di conferma o errore.
3. **ShowTickersUser**
    - **Descrizione**: Permette di visualizzare tutti i ticker associati all'utente introducendo la visualizzazione dei relativi valori di max e min, senza dover fornire manualmente l'e-mail poiché l'utente è già autenticato. Il sistema restituisce i ticker associati all'utente e un messaggio che indica se l'operazione è stata completata con successo.
4. **UpdateUser**
    - **Descrizione**: Permette all'utente di aggiornare il proprio ticker e introdotta la possibilità di inserire dei valori di max e min per il ticker che si modifica. L'e-mail

non deve essere fornita, poiché l'utente è già autenticato. Il sistema aggiorna il ticker nel database e restituisce un messaggio di successo o errore.

5. **UpdateMinMaxValue(aggiunta nell'homework 2)**
    - **Descrizione**: Permette di aggiornare i valori massimo e minimo di un ticker.Il sistema aggiorna i valori nel database e restituisce un messaggio di conferma o errore e il ticker aggiornato.

API	Descrizione	Input	Output
RegisterUser	Registra un nuovo utente e associa un ticker con valori massimo e minimo.	E-mail, ticker, max_value, min_value	Messaggio, valore boolean in caso di conferma o errore
AddTickerUtente	Aggiunge un ticker all'utente con valori massimo e minimo.	Ticker, max_value, min_value	Messaggio, valore boolean in caso di conferma o errore
UpdateUser	Aggiorna un ticker esistente con un nuovo ticker e nuovi valori massimo e minimo.	Vecchio ticker, Nuovo ticker, max_value, min_value	Messaggio, valore boolean in caso di conferma o errore
UpdateMinMaxValue	Aggiorna i valori massimo e minimo di un ticker esistente.	Ticker, max_value, min_value	Messaggio, valore boolean, ticker aggiornato

Modifica tabella UserTickers

UserTickers

Viene modificata la tabella **UserTickers**, che gestisce la relazione tra gli utenti e i loro ticker preferiti. Sono stati aggiunti due nuovi campi:

Nome	Tipo di Dato	Chiave	Relazione	Descrizione
user	VARCHAR(255)	Primary key	Collegato a **Users.email** (ON DELETE CASCADE)	E-mail dell'utente che possiede il ticker.
ticker	VARCHAR(10)	Primary key	Collegato a **Tickers.ticker** (ON DELETE CASCADE)	E-mail dell'utente che possiede il ticker.
Max_value	FLOAT			Soglia massima del ticker da controllare
Min_value	FLOAT			Soglia minima del ticker da controllare

Tabella 1 UserTickers

Documento redatto da:

Massimiliano Finocchiaro, Dario Rovito